HW2

Philip Tierney | Jared Vazzana 9/26/2023

Part 1: BSP

Code: https://github.com/Digiphoenix22/CSHW2
(MAKE SURE THAT YOUR IDE/VSCODE IS IN THE SAME FOLDER AS THE IMAGE!)

Purpose:

The code aims to create a Binary Space Partitioning (BSP) tree from a given image, and then use this tree to determine if a trace line (represented by start and end points) collides with any solid areas within the image.

Steps:

1. Image Processing:

- Load an image and convert it to grayscale.
- Detect the edges in the image using an edge detection technique.
- Identify straight lines present in the image using the Hough transform.

2. Tree Construction:

Define a basic structure (TreeNode) for nodes in our BSP tree. Each node represents a partitioning line and potentially has two children (front and back).

Ask the user to select a starting line from the detected lines in the image.

3. Recursively build the BSP tree:

- For a given partitioning line, classify remaining lines as either lying in front or behind it based on their positions.
- For the front and back groups, recursively pick a new partitioning line and classify the remaining lines again until all lines have been used or categorized.

4. Tree Visualization:

- Display the in-order traversal of the constructed BSP tree, helping to visualize the structure of the tree.

Collision Detection (bonus):

- Define a trace line using start and end points.
- Check if this trace line collides with any partitions/solid areas in the tree.
- Starting from the root of the BSP tree, determine if both start and end points of the trace line lie on the same side of the partitioning line.
- If they do, move to the relevant child (either front or back) and repeat the process.
- If they don't, it indicates the trace line crosses the partition. In this case, check both sides of the partition for potential collisions.
- If the trace line reaches a solid leaf without crossing any partitions, a collision is detected. If it reaches an empty leaf or goes through the entire tree without hitting a solid leaf, there's no collision.

Outcome:

The program will inform the user if the trace line collides with any solid areas in the image based on the BSP tree representation.

Part 2: Rotating Strings

Solution to Hackerrank problem via Clojure:

```
(defn rotations [s]
  (let [n (count s)
        concatenated-string (str s s)]
    (map (fn [i] (subs concatenated-string (+ i 1) (+ i n 1))) (range n))))

(defn main []
  (let [t (Integer. (read-line))] ; Number of test cases
```

```
        (doseq [_ (range t)]
          (let [s (read-line)
                rots (rotations s)]
            (println (clojure.string/join " " rots))))))

(main)
```

Solution to problem with Java:

```
import java.util.Scanner;

public class RotateString {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int t = scanner.nextInt();
        scanner.nextLine();  // Consume newline

        for (int i = 0; i < t; i++) {
            String s = scanner.nextLine();
            printRotations(s);
        }
    }

    public static void printRotations(String s) {
        for (int i = 1; i <= s.length(); i++) {
            System.out.print(s.substring(i) + s.substring(0, i) + " ");
        }
        System.out.println();
    }
}
```

Pseudo code:

Function main:

Read t (number of test cases)

For i from 1 to t:

Read string s

Call printRotations(s)

Function printRotations(string s):

For i from 1 to length of s:

Print s starting from i concatenated with s up to i

Print newline

Why algorithm is correct:

The algorithm takes a string and prints its rotations. The idea is to "cut" the string at every possible position and "swap" the two parts. For instance, for the string "abc", the first rotation cuts between "a" and "bc" to produce "bca". The second rotation cuts between "ab" and "c" to produce "cab". The final rotation doesn't cut at all and just produces the original string "abc".

By iterating over every position in the string and performing this "cut and swap", we generate all possible rotations. Since we handle every position and the string operations are reliable, the algorithm is correct.

4. Runtime Analysis

Let's analyze the time complexity:

- Reading the string takes time O(n), where "n" is the length of the string.
- For each rotation, we perform two substring operations. Each of these operations takes time O(n). Since we perform this operation "n" times (once for each rotation), the total time for all rotations of one string is O(n times n) or O(n^2).

For a single string, our algorithm runs in time O(n^2). If we have "t" test cases, the worst-case time complexity is O(t * n^2). However, since each string can have a different length, it's more accurate to say the time complexity is O(n^2) for each string.

The space used by our algorithm is O(n), because we only store the original string and its rotations temporarily.

Part 3: ID Search

Alright, let's tackle this problem. Given the constraints, we can use the Boyer-Moore Voting Algorithm(Source: https://medium.com/@tomas.svojanovsky11/boyer-moore-algorithm-mastering-efficient-string-searching-51312fe5098b . It's a linear time algorithm that's perfect for this problem.

Algorithm:

1. Initialize a candidate ID as None and a count as 0.
2. Traverse through the list of IDs:
   - If the count is 0, set the current ID as the candidate.
   - If the current ID is the same as the candidate, increment the count.
   - Otherwise, decrement the count.
3. The candidate ID at the end of the traversal is the ID that appears the most.

Pseudocode:

```
function findMajorityID(A: list of IDs) -> ID:
    candidate = None
    count = 0

    for id in A:
        if count == 0:
            candidate = id
        if id == candidate:
            count += 1
        else:
            count -= 1

    return candidate
```

Explanation:

The algorithm works by maintaining a count of the current candidate ID. If a new ID is encountered, the count is decremented. If the count reaches 0, a new candidate is chosen. Given the constraint that there exists an ID that appears more than 50% of the times, this algorithm will always result in that ID being the final candidate.

Why it's correct:

The key insight is that if there's an ID that appears more than 50% of the time, then it will have a net positive count after traversing the entire list. Every time another ID appears, it decrements the count of the majority ID, but since the majority ID appears more than all other IDs combined, its count will never reach zero after its first occurrence, ensuring it's selected as the candidate.

Efficiency:

The algorithm runs in O(n) time, where n is the number of IDs in the list, making it highly efficient. The space complexity is O(1) since it uses a constant amount of space regardless of the input size.